AN EMPIRICAL STUDY OF SOFTWARE ERROR DETECTION USING SELF-CHECKS*

Sung D. Cha
Nancy G. Leveson
Timothy J. Shimeall

Dept. of Information & Computer Science
University of California, Irvine
Irvine, CA 92717

John C. Knight

Dept. of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

This paper presents the results of an empirical study of error detection using self-checks. A total of twenty-four graduate students in computer science at the University of Virginia and the University of California, Irvine, were hired as programmers. Working independently, each first prepared a set of self-checks using just the specification for an application, and then each modified an existing implementation of the specification. The modified programs were analyzed to classify the various checks that the programmers wrote, and then tested to measure the error-detection performance of the checks.

The goal of this study was not just to obtain quantitative results but to learn more about such checks and how they might best be implemented. This information may result in better methods for formulating checks, making them easier to write and more effective. The analysis of the checks revealed that there are great differences in the ability of individual programmers to design effective checks. We found that some checks that might have been effective failed to detect a fault because they were badly placed, and there were numerous instances of checks signaling non-existent errors. In general, specification-based checks alone were not as effective as combining them with code-based checks. Faults were detected by the self-checks that had not been detected previously by voting 28 versions of the program over a million randomly-generated test cases.

Introduction

Crucial digital systems can fail because of faults in either software or hardware. A great deal of research in hardware design has yielded computer architectures of potentially very high reliability, such as SIFT[23] and FTMP[13]. In addition, distributed systems (incorporating fail-stop processors[20]) can provide graceful degradation and safe operation even when individual computers fail or are physically damaged.

The state of the art in software development is not as advanced. Current production methods do not yield software with the required reliability for crucial systems, and advanced methods of formal verification[12] and synthesis[18] are not able to deal with software of the required size and complexity. Fault tolerance[19] has been proposed as a technique to allow software to cope with its own faults in a manner reminiscent of the techniques employed in hardware fault tolerance. Many detailed proposals have been made in the literature, but there is little empirical evidence to judge which techniques are most effective or even whether they can be applied successfully to real problems. This study is part of an ongoing effort by the authors to collect and examine empirical data on software fault tolerance methods in order to focus future research efforts and to allow decisions to be made about real projects.

Previous studies by the authors have looked at *N*-version programming in terms of independence of failures[14], reliability improvement, and error detection[15]. Other empirical studies of *N*-version programming have been reported[5,6,9,10,11,21]. A study by Anderson[1] showed promise for recovery blocks but concluded that acceptance tests are difficult to write. Acceptance tests are a subset of the more general run-time assertion or self-check used in exception handling and testing schemes; such tests are performed after completion of a program and are essentially external tests that cannot access any local state. Such tests can, of course, also be applied to subprograms within programs. More information about the use of self-checks to detect software errors might result in better methods for formulating checks, making them easier to write and more effective. Our goal in this study was not merely to provide numerical data but to learn more about such checks and how they might best be implemented.

In order to eliminate as many independent variables from the experiment as possible, it was decided to focus on error detection apart from other issues such as recovery. This also means that the results have implications beyond software fault tolerance alone, for example in the use of embedded assertions to detect software errors during testing [22,3]. Furthermore, in some safety-critical systems (e.g., the Boeing 737-300 and the Airbus A310) error detection is the *only* objective. In these systems, software recovery is not attempted and, instead, a non-digital backup system such as an analog or human alternative is immediately given control in the event of a computer system failure. The results of this study may have immediate application in these areas. The next section describes the design of the study. Following this, the results are described and conclusions drawn.

Experimental Design

This study uses the programs developed for a previous experiment by[14]. Twenty-seven versions of a program to read radar data and determine whether an interceptor should be launched to shoot down the object (hereafter referred to as the Launch Interceptor Program, or LIP) were prepared from a common specification by graduate students and seniors at the University of Virginia and the University of California, Irvine. Extensive efforts were made to ensure that individual students did not cooperate or exchange information about their program designs during the development phase. The twenty-seven LIP programs have been analyzed by running one million randomly generated test cases on each program and locating the individual faults that were detected during the testing procedure.

*This work was supported in part by NASA under grant numbers NAG-1-511, and NAG-1-668, in part by NSF grants DCR-8406532 and DCR-8521395, and in part by MICRO grants cofunded by the state of California, Hughes Aircraft Co., and TRW.

In the present study, 8 students from UCI and 16 students from UVA were employed for a week's time to instrument the programs with self-checking code in an attempt to detect errors in the programs. Eight programs were selected from the 27 and each was randomly assigned to three students (one from UCI and two from UVA). The selection of programs to be used was accomplished by first eliminating the programs that had no previously detected faults and then randomly selecting 8 of the remaining 21 programs. The students were all graduate students in computer science with an average of 2.35 years of graduate study. Professional experience ranged from 0 to 9 years with an average of 1.7 years. None of the participants had prior knowledge of the LIP program nor were they familiar with the results of the previous experiment. There was no significant correlation found between a participant's graduate or industrial experience and their success at writing self-checks.

Participants were provided with a brief explanation of the study along with an introduction to writing self-checks. All also read Chapter 5 on Error Detection from a textbook on fault tolerance[2]. The participants were first asked to study the LIP specification and to write checks using only the specification, the training materials, and any additional references the participants desired. When they had submitted their initial checks, they were randomly assigned a program to instrument. The participants were asked to write checks with and without looking at the code in order to determine if there was a difference in effectiveness between self-checks designed by a person working from the requirements alone and those for which the person has access to and information about the program code. On the one hand, the person working only from the requirements might provide more independence by not being influenced by the written code. However, it could also be argued that looking at the code will suggest different and perhaps better self-checks. Because we anticipated that the process of examining the code might result in the participants detecting faults through code-reading alone, participants were asked to report any such detected faults but to still attempt to write a self-check to detect the fault.

The instrumented versions were subjected to an acceptability criterion (200 randomly generated test cases) as in the previous experiment. The original versions were known to run correctly on that data, and we wanted to attempt to remove obvious faults introduced by the self-checks. If any false alarms were raised by these 200 test cases (faults reported that did not actually exist) or if new faults were detected that had been introduced into the program by the instrumentation, the programs were returned to the participants for correction. Along with the instrumented version, participants submitted time sheets, background profile questionnaires, and descriptions of all program faults identified by code reading.

After the instrumented programs had satisfied the acceptability criterion, they were executed using the test cases on which they had failed in the previous experiment along with 20,000 new randomly-generated test cases to see if new faults might have been detected. Finally, the self-checks were carefully examined and catalogued as to type of check and effectiveness.

Results

The first task of the experiment participants was to read through the program requirements specification and to design self-checks based solely on that specification. These self-checks were found to fall into four groups based on the general strategy of check used:

[1] *Duplication Checks*: self-checks that duplicate the functionality of the code and compare results. Most, but not all, of the self-checks in this group use algorithms different from the original source code.

[2] *Structural Checks*: self-checks that verify the proper use of data structures or the proper semantics of code. Examples include a check that verifies that the exit condition of a loop is true immediately following the loop and a check that verifies that data values have not been improperly overwritten.

[3] *Reversal Checks*: self-checks that reverse the operation performed by the code and then see if the results are consistent with the input data.

[4] *Consistency Checks*: self-checks that determine if the results have certain properties. Examples of consistency checks include range checking, arithmetic exception checking, and type checking.

Table 1 shows the classification of the self-checks designed from the specification.[†] The participants labeled 3c and 8b did not provide specification-based self-checks. Note that the largest number of checks written were consistency checks followed by duplication checks. Performance is discussed later, but Tables 3 and 5 show that a total of 33 self-checks were completely or partially effective in detecting errors. Of these 33 effective checks, 4 (or 12%) were formulated by the participants after looking at the requirements specification only. The remaining 88% of the effective checks were designed after the participants had looked at the code. Although it might be hypothesized that acceptance tests in the recovery block structure should be based on the specification alone, our results indicate that effectiveness of the self-checks can be improved when the specification-based checks are refined and expanded by source code reading and a thorough and systematic instrumentation of the program. It appears that it is very useful for the instrumentor to actually see the code when writing self-checks.

#	Type of checks used					Total
	Dup.	Struct.	Rev.	Con.	Other*	
3a	1	0	0	10	0	11
3b	1	1	2	10	0	14
6a	2	0	0	0	0	2
6b	0	0	15	9	0	24
6c	0	0	0	14	0	14
8a	20	0	5	0	0	25
8c	15	0	15	16	0	46
12a	16	0	0	0	0	16
12b	0	1	0	26	0	27
12c	1	3	0	0	0	4
14a	16	0	1	0	0	17
14b	21	16	16	36	0	89
14c	2	0	0	4	0	6
20a	0	0	0	4	7	11
20b	15	0	4	34	2	55
20c	8	2	0	5	0	15
23a	17	0	0	0	0	17
23b	0	0	0	27	0	27
23c	0	0	18	15	0	33
25a	15	0	0	0	0	15
25b	0	0	0	8	2	10
25c	0	0	0	5	0	5
Total	149	23	76	218	11	477

Table 1: Specification-Based Self-Checks

[†] In order to aid the reader in referring to previously published descriptions of the faults found in the original LIP programs, the programs are referred to in this paper by the numbers previously assigned in the original experiment. A single letter suffix is added (a, b, or c) to distinguish between the three independent instrumentations of the programs.

*These self-checks were too vague to be classified

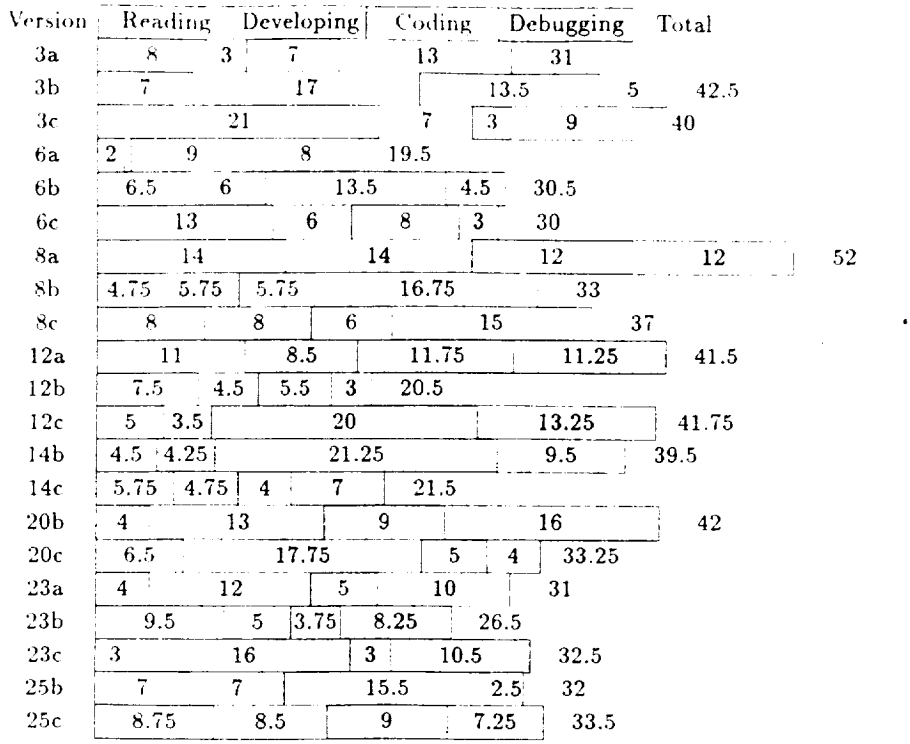

Figure 1: Summary of Participant Time-Sheets

The second task of the participants was to instrument a particular program with self-checks. No limitations were placed on the participants as to how much time could be spent (although they were paid only for a 40 hour week which effectively set an upper bound[†]) or how much code could be added. Table 2 describes the change in length in each program during instrumentation. Note that there is a great variation in the amount of code added, ranging from 48 lines to 835 lines. Participants added an average of 37 self-checks, varying from 11 to 97. Despite this variation, there was no correlation between the total number of checks inserted by a participant and the number of those checks that were effective at finding faults. That is, more checks did not necessarily mean better fault detection.

There was also no statistically significant relationship between the number of hours claimed to have been spent (as reported on the timesheets) by the participants and whether or not they detected any program faults. Figure 1 shows the amount of time each participant spent reading the specification and code, developing self-checks based on that reading, implementing the self-checks and debugging the self-checks. Three participants (14a, 20a, and 25a) did not submit a time-sheet and are excluded from this figure.

Table 3 classifies the program-based self-checks in terms of strategy used and effectiveness. Checks are classified as effective if they correctly report the presence of an error during execution. Two partially effective checks by participant 23a that detect an error most (but not all) of the time are counted as effective. Ineffective checks are those that do not signal an error when one occurs during run-time in the module being checked. False alarms signal an error when no error is present. Finally, the effectiveness is classified as unknown if the check does not signal an error and the module being tested is correct.

[†]Several reported spending more than 40 hours on the project.

Version #	Number of Lines original	a	b	c	Increase a	b	c
3	757	909	1152	805	152	395	48
6	643	859	887	700	216	244	57
8	600	1046	1356	824	446	756	224
12	573	1121	696	806	548	123	233
14	605	905	1342	712	300	737	107
20	533	611	1368	596	78	835	63
23	349	1065	417	544	716	68	195
25	906	1644	1016	1022	738	110	116

Table 2: Lines of Code Added During Instrumentation

It can be seen from the last (Total) row in Table 3 that duplication and consistency checks were about equally effective in detecting faults although more consistency checks were used. For these programs, structural and reversal checks were not effective, but this may have been influenced by the types of faults that were actually in the programs. We examined the ineffective self-checks (checks on code that contained faults but did not detect the faults) in detail. They appear to fail due to one or more of the following reasons:

- Wrong self-check strategy – the participant used a type of self-check inappropriate to detect the fault present in the code. For example, use of a structural check when the fault was an inadvertent substitution of one variable for another in an expression.

#	Effectives D	S	R	C	Ineffectives D	S	R	C	False Alarms D	S	R	C	Unknowns D	S	R	C	Total
3a	1				1	2							2	19		8	33
3b					2								11	10		13	36
3c						3								11			14
6a	2			1	3	2							6	19	1		34
6b						12		16		1				19		34	82
6c								9							2	8	19
8a	2												13				15
8b					1	4				1			6	54		2	68
8c	1				1				2			1	3	1		10	19
12a	2							2					2	3		31	40
12b								5								17	22
12c	1			8					1			1		16		8	35
14a								5						1	1	56	63
14b								3					1	1	23	37	65
14c								1							1	15	17
20a									1				4		3	3	11
20b				2	2			1	1			1	12			80	99
20c				1				1								27	29
23a	2*												9				11
23b								5								24	29
23c								2								30	32
25a	7			2												31	40
25b	1												4			6	11
25c						5								14		22	41
Total	19			14	10	28		50	5	2		3	73	168	31	462	865

Table 3: Self-Check Classification

- Wrong check placement – the participant placed the self-check in a location where not all results were checked, and the fault was on a different path.
- Use of the original faulty code in the self-check – the participant falsely assumed a portion of the code was correct and called that code as part of the self-check.

It should be noted that the placement of the checks may be as crucial as the content. This has important implications for future research in this area and for the use of self-checking in real applications.

It should not be assumed that a false alarm involved a fault in the self-checks. In fact, there were cases where an error message was printed even though both the self-check and the original code were correct. This occurred when the self-check made a calculation using a different algorithm than the original code. Because of the inaccuracies introduced by finite precision arithmetic compounded by the difference in order of operations, the self-check algorithm sometimes produced a result that differed from the original by more than the allowed tolerance. Increasing the tolerance does not necessarily solve this problem in a desirable way. This same problem occurred in our previous experiment and is discussed in detail elsewhere[7].

Some faults were detected while the participants were reading the code. The numbers in Table 4 refer to the numbering used to identify the individual faults described elsewhere[8]. Three faults were reported that actually were not faults; the participant misunderstood the code.

Table 5 summarizes the detected faults by how they were found. 20% of the detected faults were detected by specification-based checks, 40% by code-reading, and 40% by code-based checks. Note that often more than one check detected the same fault in the code-based case, which was not true of the specification-based or code-reading faults.

*These two checks were effective most, but not all, of the time.

Version	3a	6a	12c	20b	20c	25a
Fault	3.3	6.1, 6.2	12.1	20.2	20.2	25.1, 25.3

Table 4: Faults Detected Through Code-Reading

Object	Due To: Spec-based Design (SP)	Code Reading (CR)	Code-based Design (CD)	Total
Faults Detected	4	8	8	20
Effective Checks	4	8	21	33

Table 5: Fault Detection Classified by Instrumentation Technique

One final way of looking at the results of this study is to consider the number of faults detected and introduced by the participants. Table 6 shows this information.

This data makes very clear the difficulty of writing effective self-checks. Of 20 previously known faults in the programs, only 11 were detected (the 14 detected known faults in Table 6 include some multiple detections of the same fault) and only 3 of the 11 detected faults were found by more than one of the three participants instrumenting the same program. It should be noted, however, that the versions used in the experiment are highly reliable (an average of better than 99.9% success rate on the previous one million case testing), and many of the faults are quite subtle. We could find no particular types of faults that were easier to detect than others. Individual differences in ability appear to be important here.

One rather unusual case occurred. One of the new faults detected by participant 8c was detected quite by accident. There *is* a previously unknown fault in the program. However, the checking code contains the same fault. An error message is printed because the self-check code uses a different algorithm than the original, and finite precision problems cause the self-check to differ from the original by more than the allowed real-number tolerance. We discovered the new fault while evaluating the error messages printed, but it was entirely by chance. Erroneous triggering of self-checks due to finite precision problems occurred in modules that did not contain a fault, and in that case the error message was classified as a false alarm (as discussed above). Our decision was to classify the self-check as effective because it does signal a fault when a fault does exist, but this is a subjective choice.

It is very interesting that the self-checks detected 6 faults not previously detected by comparison of twenty-eight versions of the program over a million test cases[14]. After closer examination of the newly discovered faults, we found that one of the reasons they were not uncovered previously is that the random test case selection algorithm inadvertently did not allow generating those test cases that would have revealed some of the faults. This points out the well-known difficulty in selecting appropriate test cases. The fact that the self-checks uncovered new faults even though the programs were run on the same test cases that did not reveal the faults previously implies that self-checks may have advantages over voting alone. To understand why, it is instructive to examine an example of one of the previously undetected faults.

Some algorithms are unstable under a few conditions. More specifically, several mathematically valid formulae to compute the area of a triangle are not equally reliable when implemented using finite precision arithmetic. In particular, the use of Heron's formula:

#	Already Known Faults Present	Other Faults Detected SP	CR	CD	Detected SP	CR	CD	Added Faults
3a			1					
3b	4							
3c								
6a			2				1	1
6b	3							1
6c								
8a				2				
8b	2							1
8c							1	3
12a		1					1	
12b	2							2
12c			1				1	2
14a								
14b	2							4
14c								
20a								1
20b	2		1		1			2
20c			1					
23a		2						4
23b	2							
23c								
25a			2				1	
25b	3			1				1
25c								
total	20	3	8	3	1	0	5	22

Table 6: Summary of Fault Detection

$$area = \sqrt{s*(s-a)*(s-b)*(s-c)}$$

where a, b, and c are the distances between the three points and s is $(a+b+c)/2$, fails in the rare case when all the following conditions are met simultaneously:

- Three points are almost co-linear (but not exactly). s will then be extremely close to one of the distances, say a, so that $(s-a)$ will introduce round-off errors (around 10^{-16} in the hardware employed in this experiment).
- The product of the rest of the terms, $s*(s-b)*(s-c)$, is large enough (approximately 10^4) to make rounding errors significant through multiplication (approximately 10^{-12}).
- The area formed by taking the square root is slightly larger than the real number comparison tolerance (10^{-6} in our example) so that the area is not considered zero.

Other formulas, for example

$$area = \frac{x_1y_2+x_2y_3+x_3y_1-y_1x_2-y_2x_3-y_3x_1}{2}$$

where x_i and y_i are the coordinates of the three points, did not fail because the potential roundoff errors cannot become "significant" due to the order of operations. 2 of the 6 previously unknown faults detected involved the use of Heron's formula. Because the source of the unreliability is in the order of computation and inherent in the formula, relaxing the real number comparison tolerance will not prevent this problem. The fault in Heron's formula was not detected during the previous testing because the voting procedure compared the final result *only*, whereas the self-check verified the validity of the intermediate results as well. For the few cases in which it arose, the faults did not affect the correctness of the final output. However, under different circumstances the final output would have been incorrect.

Although new faults were introduced through the self-checks, this is not very surprising. It is known that changing someone else's program is difficult and whenever new code is added to a program there is a possibility of introducing faults. All software fault tolerance methods involve adding additional code of one kind or another to the basic application program. The major causes of the new faults were an algorithmic error in a redundant computation, use of an uninitialized variable during instrumentation, logic error, use of Heron's formula, infinite loops added in instrumentation, out of bounds array reference, etc. The use of uninitialized variables occurred due to incomplete program instrumentation. A participant would declare a temporary variable to hold an intermediate value during the computation, but fail to assign a value on some path through the computation. A more rigorous acceptability criterion may have detected these faults earlier, especially those that cause an abnormal termination of the program.

Conclusions

This study was not designed to provide definitive answers to any particular questions, but instead to attempt to determine what the important questions are. This should guide us and others in the design of further experiments, in the evaluation of current proposals, and in the design of new methodologies. Some important research issues arise as a result of this study that need further study such as:

[1] There appear to be great differences in individual ability to design effective self-checks. This suggests that more training or experience might be helpful. Our participants had little of either although all were familiar with the use of pre- and post-conditions and assertions to formally verify programs. The data suggests that it might also be interesting to investigate the use of teams to instrument code.

[2] The programs were instrumented with self-checks in our study by participants who did not write the original code. It would be interesting to compare this with instrumentation by the original programmer. A reasonable argument could be made both ways. The original programmer, who presumably understands the code better, might introduce fewer new faults and might be better able to place the checks. On the other hand, separate instrumentors might be more likely to detect faults since they provide a new view of the problem. More comparative data is needed here.

[3] Placement of self-checks appeared to cause problems. Some checks that might have been effective failed to detect a fault because they were badly placed. This implies either a need for better decision-making and rules for placing checks or perhaps different software design techniques to make placement easier.

[4] Specification-based checks alone were not as effective as using them together with code-based checks. This implies that fault tolerance may be enhanced if the alternate blocks in a recovery block scheme, for example, are also augmented with self-checks along with the usual acceptance test. This may also apply to pure voting schemes. A combination of fault-tolerance techniques may be more effective than any one alone. More information is needed on how best to integrate these different proposals.

[5] The process of writing self-checks is obviously difficult. However, there may be ways to provide help with this process. For example, Leveson and Shimeall (1983) suggest that safety analysis using software fault trees (Leveson and Harvey) can be used to determine the content and the placement of the most important self-checks. Other types of application or program analysis may also be of assistance. Finally, empirical data about common fault types may be important in learning how to instrument code with self-checks.

Many promising research topics, empirical studies, and experiments are suggested by the results of this study that may lead to better procedures for software error detection.

Acknowledgements

The authors are pleased to acknowledge the efforts of the experiment participants: David W. Aha, Tom Bair, Jack Beusmans, Bryan Catron, Harry S. Delugach, Siamak Emadi, Lori Fitch, W. Andrew Frye, Joe Gresh, Randy Jones, James R. Kipps, Faith Leifman, Costa Livadas, Jerry Marco, David A. Montuori, John Palesis, Nancy Pomicter, Mary Theresa Roberson, Karen Ruhleder, Brenda Gates Spielman, Yellamraju Venkata Srinivas, Tim Strayer, Gerald Reed Taylor III, and Raymond R. Wagner, Jr.

References

[1] T. Anderson, P.A. Barrett, D.N. Halliwell, and M.R. Moulding, "An Evaluation of Software Fault Tolerance in a Practical System", *Digest of Papers FTCS-15: Fifteenth Annual Symposium on Fault-Tolerant Computing*, pp. 140-145, June 1985.

[2] T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice* Englewood Cliffs, NJ, Prentice-Hall Intl., 1981.

[3] D.M. Andrews and J.T. Benson, "An Automated Program Testing Methodology and its Implementation," *Proc. 5th Int. Conference on Software Engineering*, San Diego, CA, March 1981.

[4] A. Avizienis and L. Chen, "On the Implementation of N-version Programming for Software Fault-Tolerance During Execution", *Proceedings of COMPSAC 77* pp. 149-155, November 1977.

[5] A. Avizienis and J.P.J. Kelly, "Fault Tolerance By Design Diversity: Concepts and Experiments", *IEEE Computer Magazine* , Vol. 17, No. 8, pp. 67-80, August 1984.

[6] P. Bishop, D. Esp, M. Barnes, P. Humphreys, G. Dahll, J. Lahti, and S. Yoshimura, "Project on Diverse Software - An Experiment in Software Reliability", *Proceedings of IFAC Workshop SAFECOMP'85*, October 1985.

[7] S.S. Brilliant, J.C. Knight, and N.G. Leveson, "The Consistent Comparison Problem in N-Version Software", *Software Engineering Notes*, Vol. 12, No. 1, pp. 29-34, January 1987.

[8] S.S. Brilliant, J.C. Knight, and N.G. Leveson, "Analysis of Faults in an N-Version Software Experiment", submitted for publication, 1986b.

[9] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," *Digest of Papers FTCS-8: Eighth Annual Symposium on Fault Tolerant Computing*, Toulouse, France, pp. 3-9, June 1978.

[10] J.R. Dunham, "Software Errors in Experimental Systems Having Ultra-Reliability Requirements", *Digest of Papers FTCS-16: Sixteenth Annual Symposium on Fault-Tolerant Computing*, pp 158-164, July 1986

[11] L. Gmeiner and U. Voges, "Software Diversity in Reactor Protection System: An Experiment", *Proceedings of IFAC Workshop SAFECOMP '79* pp 75-79, 1979

[12] D. Gries, *The Science Of Programming*, Springer Verlag, 1981.

[13] A.L. Hopkins, et al., "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor For Aircraft", *Proceedings of the IEEE*, Vol. 66, pp. 1221-1239, October 1978.

[14] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming", *IEEE Transaction on Software Engineering*, pp. 96-109, January 1986a.

[15] J.C. Knight and N.G. Leveson, "An Empirical Study of Failure Probabilities in Multi-Version Software", *Digest of Papers FTCS-16: Sixteenth Annual Symposium on Fault-Tolerant Computing*, pp.165-170, July 1986b.

[16] N.G. Leveson, and P.R. Harvey, "Analyzing Software Safety", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, pp 569-579, 1983.

[17] N.G. Leveson, and T.J. Shimeall, "Safety Assertions for Process-Control Systems", *Digest of Papers FTCS-13: Thirteenth Annual Symposium on Fault-Tolerant Computing*, pp 236-240, June 1983.

[18] H. Partsch and R. Steinbruggen, "Program Transformation Systems", *ACM Computing Surveys*, Vol. 15, No. 3, September 1983.

[19] B. Randell, "System Structure for Software Fault-Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, pp. 220-232, June 1975.

[20] R.D. Schlichting and F.B. Schneider, "Fail-Stop Processors: An Approach To Designing Fault-Tolerant Computing Systems", *ACM Transactions On Computer Systems*, Vol. 1, pp. 222-238, August 1983.

[21] K.R. Scott, J.W. Gault, D.F. McAllister, and J. Wiggs, "Experimental Validation of six Fault Tolerant Software Reliability Models", *Digest of Papers FTCS-14: Fourteenth Annual Symposium on Fault-Tolerant Computing*, pp 102-107, 1984.

[22] L.G. Stucki, "New Directions in Automated Tools for Improving Software Quality", *Current Trends in Programming Methodology - Volume II: Program Validation, Prentice Hall, 1977*

[23] J.H. Wensley, et al., "SIFT, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE*, Vol. 66, pp. 1240-1254, October 1978.